

# Memory Model Design for Personal Digital Assistant Operating Systems

Barclay Osborn, Rob Lomicka, Travis Newhouse, Kaoru Hosaka  
Department of Computer Science & Engineering  
University of California, San Diego

## Abstract:

The increased usage of Personal Digital Assistants (PDAs) has led to consideration of more efficient and better performing operating system for such devices. A PDA operating system must operate in a restrictive environment, where memory, CPU performance, and the power source are extremely limited, and no secondary storage exists. We explore the ramifications of memory in such environments and propose a memory model for them. We also claim that a file system can be eliminated in a PDA operating system for performance, storage use reduction, and simplicity.

## 1. Introduction

In recent years, Personal Digital Assistants (PDA's) have become extremely popular with users whose computing requirements are dominated by mobility. Mobility implies small devices, and unfortunately this size restriction currently limits the resource capacity of these devices. This "resource-poor" attribute of PDA's has raised numerous issues and subsequent solutions about how to efficiently design a PDA operating system [8, 9, 10]. The desire is typically to maximize performance, connectivity, and flexibility within the constraints of the available hardware. Theimer, et al. describe two issues that distinguish the PDA from desktop systems, namely the small size and the interaction with many services and devices which may lead to the need to execute foreign code. Subsequently, they present operating system design implications of these issues, and pose the question: "What is the right memory model to provide in response to a desire for smallness and safety?" [8] In this paper, we propose a memory model for PDA operating systems, and discuss why we believe it to be simple and adaptable/portable.

Following a discussion of what we feel are the dominating user requirements and hardware constraints for PDAs, we discuss design issues for a memory model. Additionally, we present why we feel a traditional paged virtual memory model is appropriate for PDAs because it provides a protected address spaces, a simple model for users, and can be implemented on modern hardware with minimal overhead. Additionally, we address file system issues when RAM is the last layer of the memory hierarchy. The absence of a larger capacity backing store means RAM must efficiently satisfy all of the needs for application and data storage as well as for execution space. Finally, we outline some existing operating systems and show how they have provided solutions to the "resource-poor" problem with respect to the memory subsystem.

### 1.1 User Requirements

The PDAs we're concerned with have limited memory capacity, and are generally small enough to fit in a jacket pocket. The users have different usage patterns than that of users of desktop systems. They will typically use the device for quick input or reference, although the periods of activity may be prolonged with expanded application availability and vertical market support. Accordingly, users won't tolerate long boot times or application launch delays. To solve this, it is typical to have an execution model where applications are always running and execute in place

so they don't have to be copied into an execution domain [10].

The user will expect increasing usefulness in the ability to handle multiple types of connection protocols. Since a PDA is a "go-anywhere" type of computing device, it should also be a "connect-anywhere" type of computing device. For example, it will be useful to be able to connect a PDA to an airport network in New York to download departure and arrival times, and then in the same day connect to a shopping mall network in France, to get location and inventory information about a particular shop. Having the flexibility to connect to and communicate with a heterogeneous array of external machines and networks is a crucial requirement that must be satisfied by a PDA OS [8].

It follows that network capabilities lead to security issues. While a PDA is connected to an information server on network, the user will be concerned with the security of his local data and what programs the server may run on his PDA. Although it will certainly be helpful to allow foreign code to run on the PDA in order to provide location-specific services, it is necessary for a protection mechanism to be present. This mechanism must be able to maintain the integrity of local, and sometimes very private, data while still providing the flexibility to execute and download foreign code and data. In our discussion of the memory model, we will describe the notion of trusted and untrusted code that provides a basis for security without restricting flexibility while connected to an external network.

Finally, user software demands will continually push the PDA hardware to its limit. The amount of available solid-state memory and the absence of a reliable backing store to supplement this memory poses a key challenge to satisfying the user requirements. In this paper, we explore the different considerations that must be taken into account when designing a memory model for a PDA OS as opposed to a desktop system. We propose a memory model that enables protected address spaces while maintaining minimal overhead, mechanisms to provide safe shared memory, and the abandonment of a file system.

## 1.2 Paper Outline

Section 2 discusses important hardware constraints, including the processor, storage, power source, and network interface. Section 3 describes our memory model in terms of virtual memory, file system, shared memory, and security. Section 4 compares our model with what information we could glean of existing PDA operating systems' memory models.

## 2. Hardware Constraints

While we are focusing on the importance of RAM in the construction of an OS for PDAs, there are also other aspects of the hardware that set the environment for the OS. These must be considered when proposing a memory subsystem, as they must not require capabilities beyond what the OS and hardware can support. Below we outline some of these concerns before examining the memory system in detail.

### 2.1 CPU

We believe that the CPUs of future PDAs will be slow, relative to their desktop and laptop counterparts, but will still support some of the relatively advanced features of current processors, such as modern memory management mechanisms of segmentation over paging or multilevel page tables. Already, PDAs like the now defunct Apple Newton feature reasonably fast RISC processors (ARM 610) [7] with such capabilities, and this trend will continue with high performance low power consumption chips being an active development area for microchip manufacturers. However, there are several limiting factors unique to PDAs that will constrain selection of

chips available, namely that of power and heat.

Although such issues are already being dealt with on the laptop-scale, the challenge is even greater with even smaller devices. PDAs are expected to run much longer than a laptop despite a smaller power supply, and the internals of PDAs leave even less room for heat dissipation and ventilation, and virtually no room for the luxuries of fans and large heat sinks.

## 2.2 Power Source

Although battery technology has been improving, allowing laptops and PDAs to operate for longer and longer periods of time, we see no significant advances in palm-sized mobile power sources in the near future. The improvements made over the next few years will be welcome, however, the energy consumption of PDAs will increase due to faster (but still relatively low power consuming) processors and subsystem components, higher-density RAM, and larger, higher resolution, higher bit-depth screens. While such trends are common in the laptop world, the increase in battery life in laptops over the last few years is not that impressive. The additional endurance that battery manufacturers have managed to achieve is offset by machines that consume almost that much more, essentially achieving a balance of moderate linear increase in overall endurance. In the absence of significant advances in this arena, we predict the same balance with regards to PDAs; power will always be a scarce resource as the hardware will scale to within acceptable power consumption limits. [1, 10]

## 2.3 Network Interface

Although PDAs may eventually include a high speed physical connection, we see the major advantage of PDA's connectivity to be of the form of wireless links. We consider the power consumption of a wired link irrelevant for our purposes, for if a user must physically connect to a network in the interest of bandwidth or security, a power outlet can be just as easily provided next to these links. Such a connection is an issue of the access provider of the local area network, and depends on how much they wish to cater to their users. We concern ourselves only with wireless communication.

A major concern of wireless links is that the range a link can transmit is directly proportional to the power supplied to it. Furthermore, current wireless links are unreliable and slow relative to their "wired" counterparts. We don't see this changing significantly, primarily due to the mobility of the device, particularly since the PDA will come in contact with a very heterogeneous set of networks. There will always be areas in which a signal may be undeliverable either to or from the PDA.

The possibility for information and code exchange implies that some sort of security mechanism must be provided by the OS, as both client-server and mobile agent models are ripe grounds for compromise of the PDA, local network resources, or intermediate information.

## 2.4 Storage

We predict the primary storage of PDAs will be solid-state, as a result of their size and portability. Mechanical storage devices, such as those involved in current magnetic disk technologies, don't provide sufficient protection against shocks. Moving parts can easily get displaced, rendering data inaccessible, and mechanical movement uses a significant amount of power. We believe that there will be at least one of several forms of non-volatile RAM available.

At the very minimum, non-volatility will manifest itself as the current RAM/ROM split present in current PDAs, where a portion of the

storage is fast DRAM, and the other portion slow flash-memory, including flash-RAM, flash-ROM, mask ROM, and EEPROM. Currently, flash-memory suffer from slow write speeds, the need to erase prior to writing, and a limited lifetime per sector [9]. In the center of the spectrum, there is simple battery-backed DRAM. DRAM will continue to decrease in size and increase in performance at it's current rate of quadrupling every three years [15], eventually to a size small enough that individual banks or cards of DRAM can be backed by a small replaceable lithium battery.

However, the most promising technology is that of "magram," which utilizes small ferrite particles to retain information without a power source, but otherwise have characteristic similar to today's desktop RAM modules [11]. We believe this will become the dominant source of storage and execution space for PDAs. The non-volatility property of magram will be more important to machines that want a high degree of fault tolerance or those that do not have the capability to support extremely large and inexpensive mass storage devices, like PDAs. The magram will provide fast long term stable storage and execution space, with magram "disks" available for expansion and auxiliary storage.

These limitations and advancements have major ramifications for the PDA OS. It should be small, modular, and capable of executing code in place. The user will want to devote as much of memory towards their specific needs as possible, due to the small capacity. A major consideration for the user will be balancing of the memory usage between stable store for programs and data and programs' execution space.

### 3. Memory

We focus on mechanisms required to implement a memory model. These mechanisms can be grouped into components that play an important role in the implementation of an operating system. We describe 3 high level components that we believe are integral in providing the mechanisms of the memory subsystem of a PDA operating system: the Memory Manager, File System, and Group Manager. The Memory Manager is in charge of providing address space protection for processes. Processes communicate with the Memory Manager to allocate and deallocate shared and unshared memory. The Group Manager facilitates protection mechanisms regarding access permissions to shared memory. In this section we first outline the issues surrounding the design of a memory model in an environment lacking a backing store, and then proceed to describe feasible implementations of the supporting mechanisms.

#### 3.1 Memory Model

The memory model that we feel is appropriate provides each process with its own address space. The address space of a process is divided into fixed units of allocation, called pages. At any particular instance in time, a process has access to a given set of pages. This set of pages composes the address space of the process. Protection on the types of access by a process to the pages in its address space is provided at a per page granularity. This model is simply a traditional paged virtual memory model.

In general, virtual memory is beneficial in that it can simplify the following:

- \* providing larger address spaces than physical memory
- \* relocation of data in physical memory
- \* presenting the programmer/application with its own address space
- \* protection of data from undesired access.

Let us look at the relevance of each of benefit in the context of

PDAs. Simulating a very large address becomes very difficult on a device lacking a reliable backing store. Without a backing store to swap to, the total number of pages utilized by all applications that are executing can never exceed the number of frames (physical pages) in physical memory. Therefore, while applications may be presented with the illusion of an extremely large address space, there does exist a limit on the amount of actual data that can be stored in that address space. This amount is dictated by the physical memory resources, not the virtual address space. While such a limitation does not exist in theory for a virtual memory system with a reliable backing store, real operating systems impose such a limit through the size of the swap file stored on disk. The implementation of the memory model on a PDA needs to be aware of this, and use the knowledge to its advantage. The advantage presents itself in the form of a space-efficient organization of the structures used in virtual address translation, as we discuss below.

Another effect of not being able to swap is derived from the execution model that we adopt for PDAs. Our model of execution is that processes are always running, in order to provide quick response when users wish to begin activity. Since an application is always running and never swapped out of memory, there does not appear to be a need for relocating data in memory if we use a system of fixed size allocation units - neither for temporal nor spatial performance. All locations in memory have uniform access times, aside from writing to flash-memory devices, so performance will not be improved by moving pages around in physical memory. The write latency of flash-memory devices can be remedied via simple layout policies based on type of access for the data such that relatively static, read-only data, such as application code, be placed in this type of memory. This greatly reduces the need to perform slow writes very often, most likely only during the infrequent event of application loading. Regarding space efficiency, choosing a fixed size unit of memory allocation relieves the kernel of memory management code and the difficulties that arise from allocation, deallocation, and memory compaction - all of which lead to a need to relocate data for optimal performance and minimal external fragmentation.

Thus, the benefits of virtual memory which seem most relevant to PDA systems are the presentation of individual, uniform address spaces which ease the life of the programmer and/or compiler, and the ability to designate different types of protection on various portions of the data.

Providing separate address spaces affords two benefits. One, separate address spaces present developers with a simple, abstracted view of the machine for which they are programming. This keeps with our goal to ease programming difficulty. Second, separate address spaces are a convenient way to provide protection from other errant or malicious processes on the system. Protected address spaces are desirable especially when "untrusted," mobile code may be executed.

The ability to assign permissions on the access to data is useful to both aid the programmer in finding bugs, and provide sharing. Access permissions can help programmers detect bugs in their code by causing infamous "memory access violation" errors when an unintended access takes place. Also, the ability to set permissions on data allows applications to control the manner in which their data is shared with other applications.

A drawback to virtual memory is the overhead of the metadata for whichever technique is used. For example, in a simple paging system, page tables for a 32-bit virtual address space can easily consume multiple megabytes of memory. Given a 4 KB page size and a desire to support large amounts of physical memory, the page tables would require 4 MB per process, or more. Structures of this size can quickly gobble up all of the memory

on the high-end PDAs that exist today. For desktop systems, solutions to handling such large amounts of metadata come in the form of techniques like multilevel paging or segmentation with paging. In segmentation with paging, each address space has a small segment table containing an entry per segment. Each segment table entry refers to a small size page table which has references to physical frames. When using such a technique, the segment table must reside in memory when a process is executing, but the page tables that are not currently in use can be kept on backing store, and loaded into memory as needed. Additionally, the memory manager does not even need to store page tables for segments which are not defined because the hardware will not attempt to access them. The problem we examine is how an operating system's memory manager can provide a protected address space for applications without the convenience of backing store.

The memory model we propose is not new or inventive. But, we wish to show that the model, which is common desktop systems and understood by many, is appropriate and can be realized efficiently in the PDA environment. The key constraint is that the amount of metadata for the implementation not dominate memory usage. Since memory is the last efficient and reliable level in the storage hierarchy, no swapping is available. Hence, all data, application or metadata, must reside within the confines of memory. As described above, techniques already exist, and are used by desktop systems, to overcome this constraint. We feel that, with a minor change in the semantics, PDA operating systems can still take advantage of such space efficient virtual memory techniques to organize memory.

The main change in semantics is what takes place on a page fault. On desktop machines, a page fault resulting from a lookup in the second level page table could occur because either the page is marked invalid, meaning that the address does not contain a mapping in the address space, or the page is simply not present in memory at the current time. The latter case would result in the Memory Manager loading a page from disk and retrying the instruction. The first case would cause the Memory Manager to notify the executing application of the memory access error. For a PDA, the Memory Manager requires much less logic. There is no swap space from which to load pages. Thus, page table entries cannot be valid if they are not present. So, the PDA Memory Manager has only one case to deal with, and the case, notifying the application, is the simpler of the two because the Memory Manager need not maintain tables of where pages exist on backing store.

Page faults may also occur from lookups in the segment table or first level page table. The PDA Memory Manager can immediately pass this exception to the deserving client, knowing that the memory reference was invalid. This quick decision follows from similar logic as above because all the entries in this higher level table will either be valid, or not mapped, but never have their referent residing on backing store.

Because the Memory Manager has no backing store onto which it can swap out second level page tables and application pages, it can save lots of overhead - in terms of lookup information for metadata and code size. The amount of memory needed by the Memory Manager to execute is smaller because it does not need to maintain data structures that indicate where to find swapped out metadata. Additionally, the code size of the Memory Manager can be smaller because it requires less complex page fault handling code, and it doesn't need any of the maintenance code for the data structures which are used to find metadata on backing store.

### 3.2 Hardware Support - The MMU

We do expect that the hardware of PDAs will follow in the shadows

of desktop systems. On current desktop systems, a form of multilevel paging or segmentation with paging is a common mechanism offered by modern Memory Management Unit (MMU) hardware. So, we will rely on the assumption that a typical MMU will provide a paging system with translation from virtual page number to physical page number and verification of the type of access. It is the task of the Memory Manager to maintain the tables referenced by the MMU. The specific implementation will depend on the resources of the hardware's memory management unit, but we give two examples of implementations that can be space efficient.

For our examples we will assume a 32-bit virtual address space, and an application of that requires 220 KB of static memory and 40 to 80 KB of heap and stack space to execute. (This corresponds roughly to the size of a text editor application under B-right OS [2]) In addition, all the examples will assume a 4 KB page size.

First, let us look at a metadata required by a system that supports multilevel paging. With this technique, exactly one first level page table and some number of second level page tables are stored for each address space. The size of such table can be dictated by the hardware or programmable. Let us assume that there are 2048 entries in the first level page table, 512 entries in each second level page table, and that each page table entry is 4 bytes. Then, the first level page table occupies 8 KB, and each second level page table occupies 2 KB and is capable of mapping a 2 MB portion of the address space. Now, a particular compiler may choose to place the code, static data, and heap in a contiguous portion at the beginning of the virtual address space, and the runtime stack growing upward from the end of the address space. In this situation, the metadata requirement of the application is 12 KB - for one first level table plus 2 second level tables. A more optimal compiler might recognize that 2 MB worth of address space is plenty for this application, and choose to optimize things by placing the stack region growing up from the end of the first 2 MB of the virtual address space. Then, there would only be 10 KB of metadata for the application.

An even further optimization could be realized if the MMU supports limiting the size of the virtual address space. The Memory Manager could allocate a first level page table containing only one entry, and only one second level page table that was referred to by this entry. Hence, the metadata would occupy only 2 KB + 4 bytes. The single first level entry would map virtual addresses in the range 0 to 221-1 through the second level page table, as happened in the case of the code optimized for size. Any addresses that were greater than 221 would cause the MMU to generate a fault. Again, this requires that the MMU support some sort of specifiable virtual address space, or bounds on the virtual addresses. Otherwise the Memory Manager could not get away with allocating a single first level entry because of the possibility of the MMU overrunning the length of the table during translation of addresses greater than 221.

Next, let us examine the spatial performance of a hardware system supporting segmentation with paging. Under this system, each address space has one segment table, and a page table per segment (or valid entry in the segment table). For our example, we use a segment table containing 2048 entries, and page tables with 512 entries. Segment table entries are 8 bytes, and page table entries are 4 bytes - yielding a segment table that is 16 KB, and page tables that are 2 KB each. The maximum segment size is 2 MB. First, take the case where the compiler creates one segment for each logical region of the application code, producing a segment for code, static data, heap, and stack. The overhead is then 24 KB due to a segment table plus 4 page tables. Similar compiler optimizations as used in the multi-level paging example could possibly be performed to reduce the

metadata to 18 KB, but this would be at the expense of losing the benefits of access control to red-only or executable segments. Alternatively, if the hardware supported a programmable size to limit the number of valid segment table entries, then 4 entries could be used and the overhead reduced to 8 KB.

### 3.3 File system (Or Lack Thereof)

Normally, an OS supports one, if not many, different file system formats. In our case, however, we do not support an explicit file system at all. Since programs are always running, any data created by a program exists for the lifetime of the program. If data can't be accessed by a memory operation of some process, then no application can access the data, and the data is essentially deleted. Because read and write operations on memory are atomic and permanent, and there is no reliable backing store to support a larger storage space, the concept of a file is virtually indistinguishable from ordinary program data structures. Any data that could be stored within the machine's resources must fit into available memory, and can be referenced by memory addresses just as easily. The absence of a file system gives us several advantages:

- \* No need to replicate data inside of the valuable memory resources, since all file I/O is equivalent to memory mapped I/O with instantaneous commit. Any sort of "undo" operations can be implemented as an external buffer that's copied to the final destination, if desired.

- \* No need for reorganization of data. In a disk-bound file system, the internal representation of data would need to be reorganized by the application to be stored in a format suitable for rebuilding the data structure when the application restarted. Eliminating this reorganization saves computation time and the temporary usage of memory.

- \* No computational overhead, code complexity, or space devoted to file system metadata, caching (since memory is the last level in the memory hierarchy), or file and metadata layout.

There are three instances when the traditional concept of a file differs from the local data structures of an object which must be taken into account:

Interacting with external elements. This situation can be rectified through a "filter" for incoming and outgoing data, similar to PalmOS's "conduit" feature for backing up data to another machine [12]. Since data is normally sent as a stream or sequence of packets, the only requirement is that data that is copied to a filter is formatted in a manner that the destination host can decode into a local file format. The majority of the functionality can be implemented on a per machine basis in the end host. The PDA application must do is be able to serialize the data on the way out, and rebuild data structures on the way in.

Data lifetime longer than application lifetime. Since applications are always running, this corresponds to the deletion, replacement, or crash of an application. In the case of deletion or replacement, user data can be backed up to an external machine if that data is to be maintained; perhaps to be restored when and if the data is needed. However, we don't support the existence of data that is not referenced by any process. We don't see this as a disadvantage since applications are always executing, and must already have a set of references to data in order to operate on it. If data is shared, it will only be deleted if the reference count for the page drops to 0 (see Shared Memory, 3.4 below). In the case of replacement, if both the new and old applications can fit in memory at the same time, the data can be shared between the two. When the old application is deleted, the new application can still reference the data. When an application

crashes, it's pages in memory can be saved or discarded as policy dictates. They can be saved for recovery of data on or off the PDA, or the contents discarded. Our design supports both, or possibly other, data recovery strategies.

Variables local to a process are typically not visible to another process. We remedy this by implementing a memory sharing mechanism that allows the sharing of objects created dynamically and requested to be shared while still providing a basis for security and arbitrary sharing patterns. Thus, "files" can be used as both a means of communication as well as for pure data sharing. We provide support for concurrent read sharing, and a single writer - multiple readers model, since we perceive that PDA are likely to contain highly correlated data (personal information or specific data in vertical applications). We assume it's unlikely that two processes will actually be writing at the same time, since the PDA is an inherently single-user machine, and PDAs are not likely candidates for distributed computing networks due to their low bandwidth and poor processor performance. Multiple processes may write shared data concurrently, but we don't guarantee coherency.

### 3.4 Shared Memory

Our design includes a shared memory model supporting coherent single writer-multiple reader and multiple reader access. We expect data on a given PDA to be highly correlated, as it is a single user system usually devoted to a given purpose, such as helping an individual organize his or her life, or to collect and organize data for specialized applications. Given this, it seems likely for read sharing to occur, despite the PDA being a single user system. This is a direct result of the extended lifetimes of application processes and the absence of a file system.

Our design requires shared memory to be dynamically created; we don't allow the sharing of variables on a process's stack. This requirement simplifies our proposed implementation and doesn't tie application development to a language that requires a shared object notation. If we were to allow automatic objects to be shared, we would be faced with maintaining variables that had been popped off a process's call stack after the procedure had returned, or notifying other processes of the deallocation of the variable. Data that is not created "shared" cannot later be shared, but shared data can be created and exported as "no\_access," in anticipation of sharing it later.

We support arbitrary sharing patterns among processes through the use of "groups". A group is a set of processes that can read and/or write each others shared memory; a process may be in more than one group at a time. In order to share an object A, process P would contact the Group Manager, requesting that the specified object be shared to a particular group X with a set of permissions (such as no\_access, read or read\_write). If P is a member of group X, the Group Manager complies and updates a table listing shared objects for group X.

When another process Q wants to access a shared object of P, it sends a request to the Group Manager to gain access to object A with a set of permissions. If Q is not a member of a group X, or any other group that has access to A, the request is denied. Otherwise, the Group Manager contacts the Memory Manager requesting the page containing A to be mapped into Q's page table with the permissions equal to the intersection of the permissions specified by P and requested by Q. If the mapping is successful, the Memory Manager increments the object's reference count. Sharing is performed on the granularity of pages, which is why objects must be either shared or not shared for their lifetime; it allows all objects shared by a program to the same group with the same permissions to be

placed in the same page. If many processes share many variables with different groups and access permissions, there can be severe internal fragmentation of pages, however we don't expect this to happen frequently.

Groups are created by the user or trusted user programs, arbitrary processes are not allowed to spontaneously create or change groups. It is beyond the scope of this paper to justify our policy of group creation and movement. It is safe to say we felt if we should err here, we err on the side of caution.

Although the method of sharing data via manipulation of page table entries is not new, the groups mechanism is a primitive that higher-level programs can use to implement more complex security policies. It also allows for an efficient IPC mechanism to be built on top of it. For example, a message passing-based system could use this to minimize copying message buffers, and RPC based system could use this to optimize the procedure calls, similar to LRPC [14].

### 3.5 Security Considerations

Security of shared memory is a concern due to the probability that such resource poor machines would be utilizing the surrounding network's resources, very likely in the form of mobile agents. These mobile agents would serve good purpose not just executing in the network's domain, but also on the PDA - for example, downloading a particular agent to interact with the network's particular high-level protocols.

Although the security of mobile agents is beyond the scope of this paper (and possibly the authors), we intend to provide a basis for the implementation of a trusted security-based module to be included in the system. The key distinction is in the trust of the code to be executed - some code would be trusted, and some wouldn't be. We partition code along these lines, forming two super-groups, "Trusted" and "Untrusted," which could be further sub-grouped according to the user's desire and software requirements (see Shared Memory, 3.4 above). All groups are contained within one of these two super-groups.

The critical requirement is that any user-defined group must be homogenous with regards to the Trusted and Untrusted groups, hence memory cannot be shared across the boundary between Trusted and Untrusted. Any foreign code, be it a mobile agent or an application explicitly installed by the user, will be placed in the "Untrusted" group initially.

A security manager acts as an initial verifier when code is first downloaded, perhaps allowing untrusted code to move to the Trusted group if the verification is successful. The security manager could create a new subgroup inside one of the two super-groups to "sandbox" such code; it is up to the policy makers. We leave the method of verification and policy open to implementation, be it proof-carrying code or verification of certificates and digital signatures, or something completely different. The key issues are that no code may move from the Untrusted group to the Trusted group if it doesn't pass the security manager, the security manager is trusted and there is only one, and only the security manager (with acceptance of the user) may move code from one side to the other. We believe such primitives allow for the implementation of a robust security mechanism and policy.

### 4. Comparison with Related Work

We believe that a traditional virtual memory model is appropriate for PDAs in terms of protection, usability, flexibility, and efficient implementation. However, we have observed that many existing PDAs do not use such a straightforward or protected approach. We compare our model to the memory model of Newton OS, Windows CE, B-right OS, and PalmOS.

#### 4.1 Newton OS

Newton operating system takes into consideration some common PDA problems such as severe RAM constraints, absence of permanent secondary storage device, and small power source, and resolves them by means of microkernel architecture, memory management subsystem, and support for removable storage and I/O devices. Newton MessagePad Personal Digital Assistant, which uses Newton operating system, contains 640 KB RAM, 4 MB ROM, 20MHz ARM 610 processor, and 4AAA batteries. Since this is a diskless device with no secondary storage, RAM is allocated on as-needed basis for persistent user data.

The memory structure of Newton OS supports a single address space model, where all tasks reside in a single address space. Because of this, all tasks share a single multilevel page table, and protection must be implemented at a level above the memory hardware subsystem. Newton OS utilizes an intermediate "address-oriented service" between the task and memory to verify read/write requests. The implementation is unclear, but it appears difficult to check access permissions through a fault driven model given this structure, implying that some software mechanism must verify every load and store - a major performance bottleneck. For our memory model, we utilize the virtual memory hardware to provide page level fault-driven protection. This design allows the Memory Manager to efficiently maintain a separate address space per process. In fact, fault-driven protection has already proven its performance on desktop machines. Unfortunately, the specifications on Newton OS do not provide details of the memory requirements for metadata used by their implementation, so we cannot provide a quantitative comparison of the memory usage efficiency. But, one measure we feel is important, and that is qualitative in nature, is that of understandability. We feel our design provides a much simpler, yet equally effective, model for users. Newton's memory model is not even intuitive to the authors of this paper.

Finally, the Newton operating system uses a transactional storage model that is based on objects rather than files. It facilitates the developer's task because it allows object movement to and from a permanent storage (PCMCIA storage cards, another server, etc.) without any application code. In comparison, our design requires an application to serialize the data it wishes to store externally into a format that the application will be able to recover. This may require the use of a preexisting filter, or that the application implement it's own filter; whereas Newton OS abstracts away this detail and provides a single method used by all storage requesting applications. [7]

#### 4.2 Windows CE

Windows CE operates on platforms with no disk drive, minimum of 8 MB RAM and 4 MB to 16 MB ROM. ROM contains the operating system and built-in applications. It also has ability to execute applications in place. However, applications in both ROM and RAM may be compressed, which requires decompression before they are able to be run. To speed up the loading process, Windows CE uses on-demand paging, but the success of this is debatable, as an unofficial timing of launching Microsoft Excel for Windows CE on a HP Jornada 680 took approximately 30 seconds.

This compression and decompression process is contradictory to the memory management method we propose in this paper. Compression may save some memory space, the decompression phase would reduce the performance. Also, storing compressed copies of code in ROM actually increases the use of RAM, because a decompressed copy of the code has to be put into RAM and execute from there. Instead, our goal is to maximize the usage of all

memory, whether RAM or ROM, by reducing data replication. To this end, we do not support a file system which would only duplicate application data elsewhere in memory and require overhead for metadata. Also, we try to maximize in-place execution of code and suggest a placement policy which steers relatively static, read-only data into ROM - such as long-lived application code.

Windows CE features a flat virtual address space with 32 MB of memory reserved for each process. This introduces a software-based restriction that doesn't scale well with our projected improvements in memory technologies. Our design does not impose any inherent restrictions on the address space of each process, instead the address space is limited by the hardware virtual memory support (i.e. size of virtual addresses). The actual amount of memory a process can allocate within this space is determined by the physical memory capacity. Additionally, the optimizations we suggest requiring special hardware support do not limit the virtual address space from the programmer's point of view. Instead, they request smarter compiler are to be most effective. [13]

#### 4.3 B-right

B-right emphasizes mobility, usability, real-time response, and power consumption of the PDA. It operates on the minimum of 6 MB ROM and 4 MB RAM, 2AA batteries, and 9MHz/18MHz NEC V810 processor. B-right does not require hardware virtual memory support, although the documentation does not detail their memory and protection. We see no reason why our proposed model of multiple address spaces could not be modified to run on a system without hardware support for virtual memory through the advent of load time address translation. However, a significant portion of the protection mechanisms would be lost to the point where we feel the system would lack robustness. Both designs allows for execute in place. [2, 4]

#### 4.4 PalmOS

PalmOS demonstrates a unique memory architecture that makes an efficient usage of memory space. PalmPilot products have 512 KB to 2 MB ROM, 128 KB to 2 MB RAM, 16MHz Motorola DragonBall 68328, and require 2AA batteries. ROM contains the operating system, built-in applications, and default databases. RAM contains add-on applications, system preferences, user data, and run time storage. A memory card is divided into Low Memory Globals (part of RAM), Dynamic Heap (part of RAM), Storage Heap (part of RAM), and ROM Heap. All these heaps are controlled by a memory manager. Our focus has been on the feasibility of a mechanism rather than policy. We have discussed briefly a policy on placement of data in different types of memory technologies, but it is not complete nor as detailed as that of PalmOS. While this is a very important design decision for operating system builders, we do not see any inherent restrictions to implementing "smart" policies on top of our design.

For PalmOS, the basic memory block in the heap is called a "Chunk." Our memory system does not use a variable sized memory block but is based on a fixed page determined by the MMU. A scheme of fixed size pages eliminates the memory compaction required for optimal performance when using variable sized pages. This saves both computation and relocation costs.

PalmOS has a similar approach to data storage as our proposal. Instead of using files, it uses databases that allow storage of individual records. The records can be shared among applications, records must be allocated through API calls, and are synonymous with the dynamically allocated shared object provided by our design. The major difference is that our design enables access restrictions on shared pages where records

do not.

## 5. Conclusions

We have discussed what we see as the implications of not having a backing store. Additionally, we have proposed design issues for operating systems that operate in such an environment, specifically in the context of PDAs. As the use of such consumer devices is ultimately user-driven, the underlying architecture must be ready to provide support for a variety of applications and environments with reasonable performance, security, and preparation for the next generation of technologies. Since memory is such a fundamental and evolving resource under increasing contention in these devices, there must be careful design of a memory manager. We believe traditional hierarchically organized virtual memory techniques are appropriate for PDA operating systems for several reasons, namely the efficient protection of processes within a linear address space, a coherent, abstracted machine model for programmers, good primitives for developing secure sharing systems, the elimination of file system overhead, and their implementation requires minimal overhead. However, we would like to explore this topic further with more quantitative research and gain experience with applying these ideas to an actual system.

## 6. Acknowledgments

All of us would like to thank the Graduate Lounge for its existence and Chez Bob for its sustenance. Barclay would also like to thank, Sparky, the Invisible Wonder Mutt, for his continuing assistance and patience with Barclay's inability to grasp reality.

## References

- [1] Paul J.M. Havinga, Genrad J.M. Smit. "Minimizing Energy Consumption For Wireless Computers In Moby Dick", IEEE International Conference on Personal Wireless Communications, 1997.
- [2] Akira Matsui, Yoshio Kato. "The B-right: a mBRTON-specification OS for PDA with an Affordable Window System", Proceedings. 13th TRON Project International Symposium/TEPS '96.
- [3] Paul J.M. Havinga, Gerard J.M. Smit. "The Pocket Companion's Architecture", 1996.
- [4] H.Watanabe, N.Sasaki, K.Tanigawa, Y.Kanamori. "Development of the BTRON-BrainPad", IEEE Comp. Soc. Press. 1996.
- [5] Sape J. Mullender, Paolo Corsini, Gunnar Hartvigsen. "Moby Dick-The Mobile Digital Companion", December 1995.
- [6] Robert O'Hara. "An Architecture to Simplify Communicating Applications", Proceedings of Workshop on Mobile Computing Systems and Applications, 1995.
- [7] Robert Welland, Greg Seitz, Lieh-Wuu Wang, Landon Dyer, Tim Harrington, Daniel Culbert. "The Newton Operating System", Digest of Papers. Spring COMPCON 1994.
- [8] Marvin Theimer, Alan Demers, Brent Welch. "Operating System Issues for PDAs", Proceedings of Fourth Workshop on Workstation Operating Systems,

1993.

[9] Ramon Caceres, Fred Douglass, Kai Li, Brian Marsh. "Operating System Implications of Solid-State Mobile Computers", Proceedings of Fourth Workshop on Workstation Operating Systems, 1993.

[10] Larry Loucks, Ravi Manikundalam, Freeman L. Rawson III. "A Microkernel-Based Operating System for Personal Digital Assistants", Proceedings of Fourth Workshop on Workstation Operating Systems, 1993.

[11] W. Wayt Gibbs. "The Magnetic Attraction", Scientific American, pp17-18, May 1999.

[12] Palm Computing, Inc. "PalmOS Memory Architecture," <http://www.palm.com/devzone/docs/memarch/memarch3.htm>

[13] Microsoft Corporation. "Physical Memory Use," Windows CE Manual

[14] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. "Lightweight Remote Procedure Call," ACM 1989.

[15] J. L. Hennessy, D. A. Patterson. "Computer Organization and Design: The Hardware/Software Interface," Morgan Kaufmann Publishers, Incorporated, 1998.